VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Třídění na externím úložišti

External sorting

2010

Bc. Patrik Filip

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 23. července 2010 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. července 2010 .....

Chtěl bych poděkovat Ing. Radimu Bačovi, PhD. za odborné vedení a vstřícnou pomoc při řešení problémů.

Abstrakt

Cílem této diplomové práce je popsat problém třídění dat, jejichž velikost je větší, než velikost operační paměti počítače. Seznámit se známými algoritmy, jejich nedostatky a následnými optimalizacemi, dále navrhnout komponentu pro třídění dat s použitím těchto algoritmů. Tuto komponentu implementovat v jazyce C++ nad již implementovanou strukturou perzistentního pole a následně ji otestovat, popř. dále optimalizovat.

Klíčová slova: Třídění dat, externí třídění, třídící algoritmus, složitost algoritmu, C++

Abstract

The aim of this thesis is to describe the problem of data sorting whose size is larger than the main memory. Familiarize with known algorithms, their shortcomings and subsequent optimization, design component for sorting the data using these algorithms. This component implemented in C++ over the structure of the implemented persistent array and consequently they are tested, respectively. further optimize.

Keywords: Data sorting, external sorting, sorting algorithm, algorithm complexity, C++

Obsah

Seznam tabulek

Seznam obrázků

Seznam algoritmů

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní uspěchané době si člověk snaží všemožnými způsoby ulehčit svou práci. S obrovským boomem počítačové techniky v devadesátých letech minulého století postupně narůstalo nasazování počítačů i do dříve těžko představitelných pozic. S počítačem umí pracovat děti, počítače dnes dokonce řídí letouny. K denní rutině patří komunikovat přes email, mít přístup k bankovnímu kontu prostřednictvím internetu, vést videokonference, přenášet multimédia. Umíte si bez počítače představit život? Počitače zkrátka pronikly do životů několika miliard lidí na této planetě.

Z hlediska informatiky je to jistě pozitivní informace. A nyní si zkuste představit, kolik dat vytvoří denně, týdně, měsíčně každý uživatel pracující s počitačem. Každým nákupem v internetovém obchodě, každým odeslaný emailem. Pokud tyto data chceme dále využívat, je potřeba je dříve nebo později setřídít. Pokud se množina dat k setřídění nevejde do hlavní paměti počítače, který má daná data setřídit, je nutno použit externí třídění. To zajistí správné setřídění dat řádově větších než samotná nám dostupná část hlavní paměti.

1.1 Cíl diplomové práce

Cílem mé diplomové práce je popsat současný stav známých algoritmů pro třídění dat na externím úložišti a jejich optimalizace. Dále se seznámit s již implementovanými třídami perzistentního pole, popsat jejich strukturu, chování a předvést na nich několik základních operací.

Pro tyto třídy navrhnout obecnou komponentu pro třídění využívající pro tento případ co nejvhodnější třídící algoritmus. Dále navrhnout a popsat vhodnou optimalizaci, která bude posléze také implementována. Poté popsat výslednou implementaci a v závěru provést několik srovnávacích testů a ukázku práce s výslednou aplikací.

2 Úvod do třídících algoritmů

2.1 Algoritmus

Algoritmus je předpis, který se skládá z konečného počtu kroků a který zajistí, že na základě vstupních dat budou poskytnuta data výstupní. Každý algoritmus má následující vlastnosti:

- Konečnost - požadovaný výsledek musí být poskytnut v rozumném čase. Za rozumný lze považovat čas, po kterém bude výsledek výpočtu relevantní vzhledem ke vstupním datům.
- Hromadnost – Vstupní data nejsou v popisu algoritmu reprezentována konkrétními hodnotami, ale spíše množinami, ze kterých lze data vybrat.
- Jednoznačnost – Každý předpis je složen z kroků, které na sebe navazují. Každý krok můžeme charakterizovat jako přechod z jednoho stavu algoritmu do jiného, přičemž každý stav je určen zpracovávanými daty. Vždy musí být určeno, který krok následuje.
- Opakovatelnost – Při použití stejných vstupních dat musíme dostat stejná data výstupní.
- Resultativnost – Algoritmus vede ke správnému výsledku [2].

2.2 Třídící algoritmy

Třídící algoritmus je úloha, jejímž cílem je zajistit seřazení daného souboru dat podle předem určeného klíče (numericky, abecedně, atd.) v co nejkratším čase. Problém třídění patří mezi nejvíce studované problémy v počítačové vědě, s narůstajícím trendem zaznamenávání a shromažďování digitálních údajů rostou i požadavky na třídění velkého množství dat [1].

2.2.1 Dělení třídících algoritmů

Pokud se všechna data k seřazení nevejdou do hlavní paměti počítače, probíhá *externí třídění*, kdy je k mezioperacím použito řádově pomalejší paměti, dnes převážně pevné disky. Tyto algoritmy narozdíl od klasických třídících algoritmů musí brát v úvahu mnohem větší náklady při čtení nebo zápisu na externí médium. Algoritmy pro třídění se tedy podle paměti, ve které probíhají, dělí na:

- Vnitřní – celý soubor dat k setřídění se vejde do hlavní paměti počítače, k položkám lze přistupovat náhodně se zanedbatelnými náklady.
- Vnější – do hlavní paměti počítače se vejde jen část dat k setřídění, algoritmy pro vnější třídění minimalizují počet přístupů na disk, případně řeší další optimalizace pro co možno nejlepší překrývání procesorového a vstupně/výstupního času [1].

Podle dalších vlastností pak na:

- Přirozené - přirozený algoritmus pracuje rychleji na již částečně setříděné množině dat.
- Stabilní - stabilní třídící algoritmy zachovávají relativní pořadí záznamů se stejnou hodnotou klíče. Pokud tedy v souboru dat existují dva záznamy se stejnou hodnotou klíče, musí algoritmus zachovat jejich vzájemné pořadí.

2.2.2 Složitost třídících algoritmů

- Časová - časovou složitostí rozumíme funkci, která každé množině vstupních dat přiřazuje počet operací vykonaných při výpočtu podle daného algoritmu.
- Paměťová - paměťovou složitost definujeme jako závislost paměťových nároků algoritmu na vstupních datech.

Časová složitost výpočetních metod zpravidla vzbuzuje menší respekt než složitost paměťová [2].

2.3 Základní typy třídících algoritmů

- Bubblesort - Při neustálém procházení seznamu porovnáváme dva sousední prvky, pokud nejsou ve správném pořadí, vyměníme je. Procházení seznamu skončí, pokud je celý setříděn. Jednoduchý na implementaci. Přirozený, stabilní. Vyžaduje velké množství zápisů do paměti. Průměrná i maximální složitost: $O(n^2)$.
- Quicksort - Patří mezi algoritmy typu rozděl a panuj. Nejprve na seznamu zvolíme *pivota* (nejlépe medián celého seznamu). Algoritmus pak seznam rozdělí na dva seznamy, v němž v jednom jsou prvky menší než pivot, ve druhém větší. Oba tyto seznamy se stejným postupem rekurzivně setřídí. Složitost algoritmu závisí na volbě pivota, počítat medián u velkého množství dat však není efektivní, používá se tedy náhodně vybraný prvek (nepříliš efektivní), nebo medián náhodně vybrané malé části seznamu. Nepřirozený, nestabilní. Průměrná složitost $O(n \cdot \log n)$, maximální složitost: $O(n^2)$ (při použití vždy nejhoršího možného pivota).
- Mergesort - Řazení sléváním, další z algoritmů typu rozděl a panuj. Algoritmus postupně dělí seznam na menší části, dokud není rozdělený na jednotlivé prvky (triviální problém). Pak jej postupně spojuje zpátky s přihlédnutím na hodnoty klíčů (výběr nejvhodnějšího klíče). Přirozený, stabilní.
  Průměrná i maximální složitost: $O(n \cdot \log n)$.[1]

[1] Toto jsou jen jedny z nejznámějších třídících algoritmů

3 Algoritmy pro třídění na externím úložišti

Jak jsme již uvedli dříve, třídění mimo hlavní pamět sebou přináší nové nároky na efektivitu algoritmu, jde především o problémy spojené s ukládáním na řádově pomalejší médium, pevný disk. Ten dnes představuje nejpomalejší část počítače, jeho typické vlastnosti vychází ze staré kontrukce využívající magnetickou indukci a mechanické přesouvání záznamových hlav. Naštěstí jsou dostupné plně elektronické *SSD disky*, u kterých odpadají mechanické neduhy při zápisu či čtění dat. Vyhneme se tak dosti velké přístupové době a závislosti na fyzickém umístění zapsaných dat, nemluvě o případné fragmentaci souborů [4].

3.1 Uvažovaný model

V celé diplomové práci i pozdějším řešení uvažuji následující model: Předpokládám, že počítač, na kterém bude třídění probíhat, má jednu hlavní paměť, jeden procesor a jeden disk, které budu používat pro operace potřebné k běhu algoritmu. Jako jednu stranu paměti uvažuji takovou část hlavní paměti, která má stejnou velikost, jako alokační jednotka používaného souborového systému.

Převážná většina algoritmů pro vnitřní třídění nemůže být upravena tak, aby efektivně pracovala jako algoritmus pro vnější třídění. Metody využívající sekvenčního přístupu jako je např. insertsort a selectsort můžeme z dalších úvah rovnou vyloučit. V milulosti byly algoritmy pro vnitřní třídění studovány také v souvislostí s virtuální pamětí, výsledky však nebyly tak slibné důsledkem nadměrného počtu odkládacích stránek. Snaha přepracovat heapsort a quicksort pro vnější třídění nebyla efektivní, nepodařilo se dosáhnout tak malého počtu vstupně výstupních operací jako u mergesortu, který je z tohoho hlediska nejlepším algoritmem pro vnější třídění.

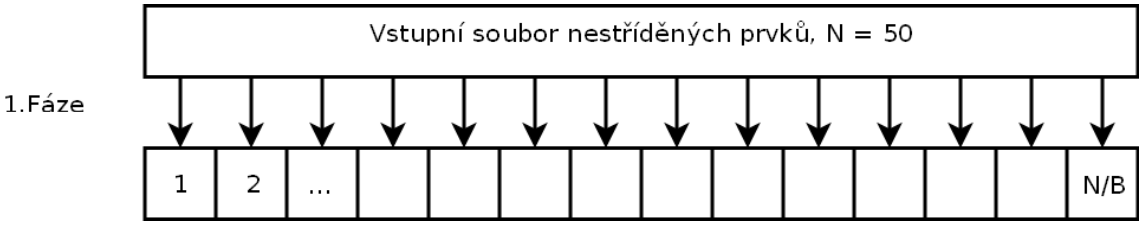
V posledních letech se výzkum zaměřuje na jeho optimalizaci vedoucí k co největšímu zefektivnění a optimálnímu překrývání procesorového a vstupně/výstupního času [1].

3.2 Popis externího mergesortu

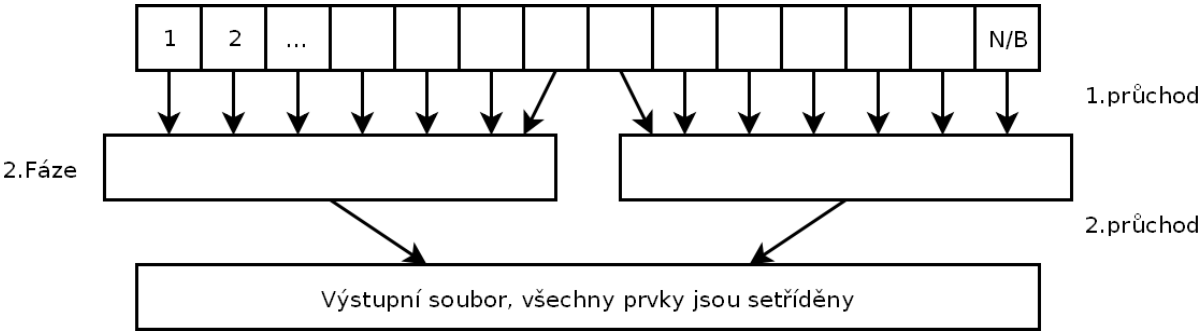
Předpokládejme, že soubor, který má být setříděn, má N stran a že máme k dispozici B stran v hlavní paměti. Třídící algoritmus pak probíhá ve dvou fázích:

- V první fázi je soubor rozdělen do částí, které se nazývají *skupiny* nebo *vlákna*. Celá skupina může být zcela uložena v B stranách hlavní paměti a tudíž existuje N/B skupin vyprodukovaných v této fázi. Každá skupina je setříděna nezávisle v hlavní paměti některým z vnitřních třídících algoritmů a pak zapsána zpět na disk, viz Obrázek 1.

- V průběhu druhé fáze jsou skupiny slévány tak, aby vyrobily menší počet skupin a nakonec byly spojeny do jediné výsledné. Spojovací fáze se může skládat z několika průchodů. Dostupnou paměť ve druhé fázi rozdělíme tak, aby vznikla jedna vyrovnávací paměť pro výstupní soubor a zbytek paměti se rozdělil jako čtecí vyrovnávací pamět jednotlivých skupin. V každém průchodu se z každé skupiny,



Obrázek 1: První fáze externího mergesortu



Obrázek 2: Druhá fáze externího mergesortu využívající dva průchody

resp. jejího čtecího zásobníku, vybere nejmenší popř. největší prvek a ten se porovnává s dalšími takto získanými prvky z ostatních skupin. Na konci každého průchodu dostaneme vždy jeden prvek, který zapíšeme na výstupní zásobník. Pokud je výstupní zásobník plný, zapíše se do výsledného souboru, obdobně, pokud je některý ze vstupních zásobníků prázdný, načte se do něj další část příslušné skupiny [3], viz Obrázek 2.

Výkonnost externího mergesortu záleží mj. na výběru spojovacího schéma, jakým se vybírají skupiny pro spojování ve druhé části algoritmu. Různé postupy totiž mohou vést ke značným výkonovým rozdílům [1].

3.2.1 Algoritmy pro tvorbu skupin

Pro tvorbu skupin se běžně používají dva algoritmy:

První, nazývající se *načti-setřiď-ulož*, naplní všechny dostupné stránky hlavní paměti záznamy ze souboru k seřazení a spustí vnitřní třídicí algoritmus heapsort nebo quicksort. Výsledek je zapsán zpět na disk jako setříděná skupina. Tento postup se opakuje, dokud nejsou setříděny všechny záznamy. Jednotlivé skupiny tedy mají velikost shodnou s velikostí dostupné hlavní paměti.

Druhý algoritmus se nazývá *výměnný výběr*. Využívá minimální haldu k výběru klíče s minimální hodnotou. Halda se nejprve naplní, pak je záznam s nejmenší hodnotou klíče (vrchol haldy) přesunut z haldy do výstupního zásobníku a tím uvolní místo pro nový prvek. Další prvek, který má být přesunut do výstupního zásobníku, musí mít větší

Algoritmus 1: Ukázka pseudokódu externího mergesortu

```
Input: file
BLOCKSIZE;
RAMSIZE;
N=size(file)/BLOCKSIZE;
B=RAMSIZE/BLOCKSIZE;
//první část
while file != empty do
    for i:=1 to N/B do
        ram:=load(file)/RAMSIZE;
        sort(ram);
        write(filesort, i);
// druhá část
for b:=1 to B-1 do
    alokuj(B_i, B/ N+1);
alokuj(B_out, B/ N+1);
for b:=1 to N/B do
    B_b:=read(filesort, b, B/ N+1);
while filesort != empty do
    B_out = merge(B_1 ... B_N/B);
    if B_x is empty then
        doplň(filesort, x + přečteno);
    if B_out is full then
        write(B_out, finallyfile);
        B_out.clear();
```

klíčovou hodnotu, než poslední vyjmutý prvek. Pokud se v souboru vyskytnou záznamy s menší klíčovou hodnotou, než poslední prvek vložený do výstupního zásobníku, umístí se do nové minimální haldy. Tím se postupně první halda zmenšuje a druhá zvětšuje. Při vyprázdnění první haldy je dokončena tvorba první skupiny a prvky z další haldy již plní další skupinu. Tento proces končí, když v souboru nezbývají žádné další záznamy k setřídění [3].

Jak bylo prokázáno [3], průměrný počet skupin vytvořených výměnným výběrem je poloviční oproti velikosti skupin, vytvořených algoritmem načti-setřiď-ulož. To má vliv na výpočetní čas druhé části algoritmu, menší počet skupin totiž zkrátí čas potřebný k vykonání druhé části externího mergesortu. Algoritmus výměnný výběr má stálejší průběh vstupně/výstupních operací a dosahuje lepších výsledků na již částečně předtříděných datech. Mezi jeho nevýhody patří nutnost současného čtení záznamů ze vstupního zásobníku a zápis na výstupní zásobník. Jsou-li oba soubory uloženy na stejném disku, je potřeba více času na režii při čtení a zápisu dat, vyplývající z těchto požadavků. Pokud ovšem máme k dispozici dva disky, je výměnný výběr lepší volbou [1].

3.2.2 Správa paměti pro záznamy různých délek

V reálném světě potřebujeme pracovat se záznamy různých délek. Výše popsané algoritmy však nejsou přímo použitelné pro záznamy s různými délkami, protože paměťový prostor pro třídění je pevně dán a počet záznamů, které mohou být vyjmuty z tohoto prostoru, není předem znám. Budeme předpokládat, že se paměťový prostor skládá z několika rozsahů, protože není možné alokovat libovolně velké části sousedící paměti. Postup je následující: Při spuštění algoritmu pro tvorbu skupin jsou záznamy s proměnnou délkou čteny ze vstupu a vkládány do paměťového prostoru. Tato metoda má najít místo pro každý nový záznam uvnitř existujícího paměťového prostoru. Pokud není nalezeno žádné volné místo, je záznam přesunut z paměťového prostoru na výstup a tím vytvoří prostor pro nový vstup. Pokud záznamy neobsadí všechna vytvořená volná místa, musí pak systém sledovat volné segmenty a kontrolovat možnost sloučení potencionálních sousedních volných segmentů. V následujících dvou heuristických metodách je popsáno hledání volných segmentů.

Další vyhovující - tato metoda sekvenčně vyhledává všechny volné segmenty paměti, začíná od pozice, kde došlo k poslednímu vložení, a probíhá, dokud nenalezne dostatečně velkou část paměti pro uložení nového záznamu. Jelikož může být vyhledávání dosti náročné, zavádí se prahová hodnota k omezení počtu zkoumaných volných segmentů. Pokud je tento limit dosažen a nebyl nalezen žádný volný segment, snaží se algoritmus přesunout záznam sousedící s volným segmentem z jeho aktuální pozice k dalšímu volnému segmentu, který byl zkoumán v průběhu stejného vyhledávání. Pokud tento přesun vytvoří požadované místo, je záznam vložen do paměťového prostoru, pokud nevytvoří, je jeden nebo více existujících záznamů odstraněno z paměťového prostoru a vytvoří tak vhodné volné místo.

Nejlepší vyhovující - v této metodě jsou nové záznamy vkládány do nejmenšího volného segmentu, který je natolik velký, aby daný záznam přijal. Pro efektivní vyhledávání vhodného volného segmentu sleduje algoritmus velikosti volných segmentů v binárním

stromu. Při vkládání nového záznamu prohledá algoritmus binární strom, aby nalezl volný segment minimální velikosti, který je větší nebo roven velikosti nového záznamu. V případě, že neexistuje žádný volný segment, jsou existující záznamy odstraněny, dokud není dostatek volného místa.

Podle testů, které byly provedeny, dosahuje metoda nejlepší vyhovující až 95% využití paměti pro velké paměťové rozsahy. Pro menší paměťové rozsahy je využití kolem 75%. Metoda nejlepší vyhovující je dále časově úspornější - náklady na vyhledávání v této metodě totiž narůstají s počtem volných segmentů logaritmicky (díky využití binárního stromu). Pro záznamy s proměnnou velikostí je tedy metoda nejlepší vyhovující s metodou výměnného výběru tou nejlepší kombinací [1].

3.3 Spojovací algoritmy

Vzhledem k tomu, že je spojování skupin převážně vstupně/výstupní operací, snaží se všechny algoritmy minimalizovat počet přístupů na disk. Vyvážený *k-cestný* mergesort třídí data použitím opakovaného slučování. Rozdělí vstup do dvou skupin opakovaným čtením bloků dat ze vstupu, které zaplní hlavní paměť, skupina se setřídí, a pak se zapíše do další skupiny. Ve druhé fázi se opakovaně spojují dvě skupiny do jedné ze dvou výstupních skupin, dokud nevznikne jedna setříděná skupina [7].

Dřívější studie ukázaly, u vícecestného spojování, obecně *k*-cestného spojování, zvolením *k* co nejvyššího nemusí vždy vést ke zlepšení výkonnosti. Jedno z řešení je vložit nejprve tolik prázdných skupin, kolik je potřeba k vytvoření všech *skupin-1*, dělitelných *k-1*. Pak algoritmus spojí v každém průchodu *k* nejkratších skupin, dokud nezůstane jen jedna. Tento postup však nedosahuje překrývání mezi vstupně/výstupním a CPU časem: procesor zůstane v nečinnosti, zatímco je *k* zásobníků plněno částmi odpovídajících skupin načítaných z disku. Tento problém řeší prognostická metoda, metoda dvojitého vyrovnávání nebo metoda prokládaného rozvržení.

Prognostická metoda - tato metoda sleduje čtecí zásobník, který bude vyprázdněn jako první a používá další zvláštní zásobník ke čtení další vhodné části z disku, dokud obsah zbývajících zásobníků nepokračuje ke zpracování. *Vylepšená prognostická metoda* zavádí posloupnost spotřeby k určení, který ze zásobníků bude vyprázdněn jako první a čtení dat z disku začíná již ve chvíli, kdy je v paměti volných *m* stran. Nemusíme tedy čekat na uvolnění celého zásobníku některé skupiny [1].

Dvojité vyrovnávání - v této metodě se pro každou skupinu vytvoří dva zásobníky, jeden pro provádění V/V operací a druhý pro zpracování. Může být použita buď při čtení z disku nebo při zápisu na něj. Oproti prognostické metodě vyžaduje tato metoda více paměti, nebo snížení velikosti stávajícího zásobníku. Díky dvěma zásobníkům může být dosaženo překrývání mezi V/V a CPU časem, ale nutno poznamenat, že nemá využití u téměř setříděných dat [5]. V rozšíření této metody, nazývajícím se *rovnoměrné vyrovnávání*, se pro každou skupinu vytvoří *m* zásobníků. Nyní nastává otázka, jak brzy zahájit nové čtení pro zaplnění prázdných zásobníků. Pokud začneme číst, když je prázdných *m-1* zásobníků, minimalizujeme vyhledávací čas, neboť všech *m-1* bloků bude čteno z po sobě jdoucích míst. Tím však není zaručeno úplné překrytí CPU a V/V času, protože při čtení *m-1* bloků může CPU zůstat nečinný. V tomto okamžiku se využívá vlastnosti

moderních diskových systémů, které rozdělí čtení velkého množství dat na čtení posloupnosi několika menších datových bloků. V průběhu plnění zásobníků sekvenčním čtením z disku, jsou již některé zásobníky naplněny a mohou být použity k dalšímu zpracování. Nevýhodou je, že čtení nemůže začít dříve, dokud není prázdných *m-1* zásobníků patřících ke stejným skupinám, operační paměť tedy není plně využita.

V průběhu slučovací fáze jsou data čtena v odlišném pořadí, než v jakém byla zapisována ve fázi tvorby skupin. Ve fázi pro tvorbu skupin se totiž skupiny zapisují na po sobě jdoucí místa na disku, narozdíl od spojovací fáze, kde jsou data čtena v určitém pořadí, které závisí na pořadí vstupních záznamů. To má dopad na celkové přístupové náklady na disk, zvyšuje se totiž vyhledávací doba. Metoda *prokládané rozvržení* [6] umisťuje záznamy z různých skupin do souvislých pozic podle algoritmu robin-round. Je založena na očekávání, že místo nového záznamu je blíže k pořadí, v jakém budou čteny záznamy z disku a takto snižuje vyhledávací čas. Představuje však režii v průběhu fáze pro tvorbu skupin.

3.3.1 Plánování čtení

Každá skupina je rozdělena do bloků o délce rovnající se velikosti vyrovnávací paměti. Uložením maximální klíčové hodnoty z každého bloku dat jsme schopni určit pořadí, ve kterém budou bloky požadovány v průběhu spojovací fáze. Toto pořadí se nazývá *posloupnost spotřeby* [6]. Ukládání maximální klíčové hodnoty nepředstavuje významnou režii, protože lze provést v hlavní paměti. Tyto klíče by měly být setříděny, což je možno provést, když se začíná zapisovat poslední skupina na disk (v průběhu tvorby skupin) nebo během předcházející spojovací fáze.

Pokud jsou během spojovací fáze dostupné nějaké extra zásobníky, lze je použít pro čtení datových bloků, které nejsou aktuálně potřebné v nějaké části spojování. Čtení těchto bloků nepředstavuje rotační nebo vyhledávající zpoždění, jelikož bloky jsou v po sobě jdoucích pozicích. Posloupnost spotřeby můžeme použít k určení, které bloky budou potřebné v následujících fázích, načíst je za nízké náklady a urychlit tak spojovací část.

Tento problém lze formulovat následovně [6]: Buď n počet skupin, T počet datových bloků, B počet zásobníků. Pak $C = \{C_1, C_2, ..., C_T\}$ je blok posloupnosti spotřeby, $R = \{R_1, R_2, ..., R_T\}$ je čtecí sekvence, L je mapovací funkce mezi datovými bloky a pozicemi disku, které jsou označeny čísly 1, 2, ..., T. Každá čtecí sekvence R je charakterizována svou cenou, která je rovna součtu celkového vyhledávacího času a celkového přenosového času potřebného ke čtení datových bloků v pořadí určeném R. Vzhledem k tomu, že každá čtecí sekvence má stejou dobu přenosu, výběr nejlepší sekvence je založen na výpočtu hledacího času. Čtecí sekvence navíc musí splňovat další podmínku - v každé fázi musí být uchován nejméně jeden blok dat z každé skupiny v jednom z dostupných zásobníků. Sekvence splňující tuto podmínku se nazývá *proveditelná*. Musíme tedy najít proveditelnou sekvenci, která má minimální hledací čas.

Lze snadno odvodit, že problém nalezení optimální proveditelné čtecí sekvence je ekvivalentní s problémem obchodního cestujícího, proto může být navrženo jen heuristické řešení. Heuristika je založená na umístění každého C_i údaje z bloku posloupnosti

spotřeby *C* uvnitř čtecí sekvence *R* v pozici, která minimalizuje vyhledávací čas. Ten se vypočte podle $L(R_i)$ pozice disku, kde jsou uloženy údaje o čtecí sekvenci. Aby byla čtecí sekvence proveditelná, je udržována další sekvence *F*. *J*-tý údaj sekvence *F* představuje počet volných zásobníků před přečtením údaje R_j ve čtecí sekvenci. Pokud je C_i vloženo na pozici *p* čtecí sekvence, dostaneme R_p, kde $p < i$, pak je pro jeho uložení potřeba dalšího zásobníku. Proto je F_j zmenšeno o 1, když $p \leq j < i$, ve smyslu, že pro bloky $R_p, ..., R_{i-1}$ bude o jeden zásobník méně, než když začaly být čteny. Pokud je $F_j = 0$ pro dané *j*, pak žádný blok za *j* z posloupnosti spotřeby nemůže být umístěn před *j* ve čtecí sekvenci. Proměnná *s* zaznamenává poslední hodnotu *j*, kde je $F_j = 0$.

Algoritmus 2: Ukázka pseudokódu algoritmu posloupnosti spotřeby

```
R1:=C1;
s:=1;
for j:=1 to T do
    Fj:=B-n;
for i:=2 to T do
    p:=i;
    for j:=s to i-1 do
        if Rj se nachází na stejné stopě na disku jako Ci AND L(Ri) > L(Ci) then
            p:=j;
            break;
    for k:=i downto p+1 do
        Rk := Rk − −;
        Fk := Fk−1 − −;
        if Fk=0 AND sj=k then
            s:=k+1;
    Rp:=Ci;
```

Popsaný algoritmus má složitost $O(T^2)$ a může být spuštěn, dokud je na disku uložena poslední skupina. Metoda plánování čtení s použitím posloupnosti spotřeby dokázala překonat metody dvojitého vyrovnávání i prognostickou metodu [6].

3.3.2 Seskupování bloků

Tato metoda se snaží o nalezení heuristického řešení problému optimální čtecí cesty. Je založena na seskupení takového množství sousedních bloků dat ze stejné skupiny, jak je jen možné. Několik sousedních bloků ze stejné skupiny tvoří celek, který může být přečten sekvenčně. Čím menší je počet skupin, tím méně vyhledávacího času bude potřeba během spojování. Tato čtecí sekvence (celek) je proveditelná, pokud je v průběhu spojovací části přečten každý blok právě jednou. Metoda pro stanovení proveditelnosti čtecí sekvence je založena na počtu volných zásobníků, *F*. F_i je rovno počtu volných

zásobníkových stránek po přečtení i-tého celku. Velikosti celků jsou označeny $L = \{L_1, L_2, ..., L_N\}$ pro dané N [6].

Algoritmus 3: Ukázka pseudokódu algoritmu seskupování bloků

```
R:=C;
//čtecí sekvence se inicializuje podobně jako posloupnost spotřeby
for i:=1 to T do
    L_i:=1;
    //nastavení prvotní velikosti celku
lastC1:=n;
for i:=n+1 to T do
    for j:=lastC1 downto 1 do
        if R_j.runNumber=R_i.runNumber then
            k:=j;
            break;
    if lze R_i zkombinovat s R_K při zachování proveditelnosti then
        L_K++;
    else
        //R_i se stane novým celkem lastC1++;
        R_lastC1++;
N:=lastC1;
```

Seskupování bloků je nejefektivnější metodou ze všech variant pro zlepšení spojovacích algoritmů. Oproti metodě plánování čtení, která je druhá nejlepší, dosahuje nárůst výkonu 30%. Navíc je schopna využít již částečně předtříděná data [1].

Externí mergesort je doposud nejlepší třídící algoritmus pro vnější třídění [1], zde uvedené optimalizace ještě dále vylepšují jeho výkonnost. Obecně jde o snahu minimalizovat prodlevu při čtení a ukládání na pevný disk (či ji co nejvíce využít k další optimalizaci) nebo překrývání procesorového a vstupně/výstupního času.

4 Komponenta perzistentního pole

4.1 Popis komponenty

Při úvahami nad analýzou třídění dat na externím úložišti podle popsaných algoritmů se nám naskýtá několik problémů. Mezi jeden z nich patří jakým způsobem navrhnout a implementovat vyrovnávací paměť pro čtení a zápis souborů. Jelikož je nutno mít kontrolu nad volbou velikostí jednotlivých vyrovnávacích pamětí, je vyrovnávací paměť v souborovém systému operačního systému nevyhovující. Rozhodl jsem se využít řešení, které je již naimplementováno a všechny potřebné podmínky splňuje - perzistentní pole.

Perzistentní pole se z hlediska programátora tváří jako dynamické pole vektorů, které všechna svá data samostatně ukládá do externího souboru a napodobuje svým chováním reálný souborový systém operačního systému. Hlaví výhoda perzistentního pole narozdíl od běžného datového souboru uloženém v souborovém systému spočívá ve vlastní vyrovnávací paměť cache, která umožňuje lépe řídit, co bude uloženo na pevném disku a co v hlavní paměti. Všechny třídy implementující toto pole jsem převzal z aplikačního rámce ATOM implementovaný databázovou skupinou na Katedře informatiky Fakulty elektrotechniky a informatiky, Vysoké školy Báňské - Technické univerzity Ostrava.

Nyní již k samotným třídám implementujícím perzistentní pole. To v sobě ukládá vektory typu zděděného ze třídy *cBasicType*, v následujících příkladech i v této práci budu používat prvky, jež reprezentuje třída *cNTreeTuple_NoEncodeType*. Nezbytnou součástí je šablona *cSizeInfo*, která zajišťuje správný výpočet velikosti položky, důležitý především u složitějších typů. Parametrizuje se položkami konkrétního typu zděděné z již zmiňované třídy *cBasicType*. Pro vkládání prvků typu *cNTreeTuple* je připravená třída *cNTupleSizeInfo* parametrizující šablonu *cSizeInfo* typem *cNTreeTuple*. Tato třída obsahuje informace i pro zde používaný typ prvků *cNTreeTuple_NoEncodeType*.

4.2 Příklad práce s perzistentním polem

Pro práci se samotným perzistentním pole potřebujeme vytvořit několik pomocných objektů a definovat parametry při jejich vytváření:

- Nejprve je třeba definovat typ vkládaných prvků. V následujících příkladech budou jednotlivé prvky typu *cNTreeTuple_NoEncodeType*, jednotlivé vektory mohou mít proměnnou délku, tedy každý z nich nemusí mít délku určenou dimenzí. Poslední definicí typu *tPersistentArray* jen vložíme předchozí typy do šablony samotného perzistentního pole.

- Dále je třeba *popisovač místa*, při jehož vytváření vkládáme do kontruktoru hodnoty maximální dimenze vektorů a jejich datový typ, dědící ze třídy *cDataType*. Popisovač místa je nová instance třídy zděděné z obecného popisovače *cTreeSpaceDescriptor*. V tomto případě tedy bude maximální dimenze vektorů 10 prvků a jednotlivé prvky vektoru budou neznaménkové celočíselné, viz řádek 7 ve zdrojovém kódu 1.

- *Hlavička* perzistentního pole je zděděná ze třídy *cHeader*. Jak již název napovídá, jde o hlavičku ekvivalentní ke zvolenému typu prvku i samotného perzistentního pole, tj. *cPersistentArrayHeader_VarLen*. Jako parametry konstruktoru vkládám informace o volném místě, kterému dále předám dříve vytvořený popisovač místa. Tato informace o volném místě dědí ze základní třídy *cSizeInfo* a odpovídá zvolenému typu prvku pole. Druhý vkládaný parametr je velikost uzlu vkládaného prvku, třetí pak určuje velikost bloku, v podstatě jde velikost alokační jednotky, zde tedy 4kB.
- Nyní již můžeme vytvořit samotné *perzistentní pole*, jako parametr konstruktoru předávám dříve vytvořenou hlavičku, viz řádek 15 ve zdrojovém kódu 1.
- Pro čtení vektorů z pole používám *kontext*, sloužící k uložení informací potřebných pro práci s perzistentním polem s proměnnou délkou vektorů. Sleduje uzel, aktuální místo v uzlu a aktuální položku. Ukládá také další pomocnou paměť. Jde o třídu zděděnou ze základní třídy *cDataStructureContext*. Její typ opět odpovídá typu perzistentního pole. Jako parametr kontruktoru vkládám hlavičku, která obsahuje všechny potřebné údaje pro jeho správnou inicializaci.

Nyní chvíli máme vytvořeny a inicializovány všechny potřebné objekty a můžeme začít pracovat se samotným polem. Vstupní soubor pole načteme metodou *Open()* (řádek č. 21), kde jako parametry uvedeme název otevíraného souboru, dále logickou hodnotu, má-li být daný soubor otevřen jen pro čtení a velikost vyrovnávací paměti v počtech bloků. Další metodou *OpenContext()* (řádek č. 23) otevřu kontext daného pole na indexu a pozici, které mi určují první dva parametry. Zde je to logicky začátek celého pole. Dále v jednoduchém cyklu vypisuji všechny vektory a jejich jednotlivé prvky ze souboru perzistentního pole, dokud kontext nedojde na jeho konec. Metodou *Advance()* (řádek č. 29) posunu kontext z aktuálního vektoru vždy o jeden dále, pokud kontext dojde na konec souboru, vrací metoda logickou hodnotu false. Po ukončení práce s kontextem stačí zavolat metodu *CloseContext()* (řádek č. 31), obdobně zavoláme po skončení práce s perzistentním polem metodu *Close()* (řádek č. 32).

Nyní se již podívejme na samotný zdrojový kód:

```
1  void Ukazka::Vypis_ze_souboru(){
2
3      typedef cNTreeTuple_NoEncodeType Type;
4      typedef cPersistentArrayNode_VarLen<Type> tNode;
5      typedef cPersistentArray_VarLen<Type, tNode> tPersistentArray;
6
7      cNTreeSpaceDescriptor *popisovac = new cNTreeSpaceDescriptor(10, new cUIntType());
8
9      cPersistentArrayHeader_VarLen<Type> *hlavicka =
10         new cPersistentArrayHeader_VarLen<Type>(
11             new cNTupleSizeInfo(popisovac),
12             tNode::GetNodeExtraSize(),
13             4096);
14
15     tPersistentArray  *pole = new tPersistentArray(hlavicka);
16
17     cPersistentArrayContext_VarLen<Type> *kontext =
```

```
18          new cPersistentArrayContext_VarLen<Type>(hlavicka);
19
20
21      pole->Open("Output.dat", true, 8);
22
23      pole->OpenContext(0, 0, kontext);
24
25      do{
26
27          kontext->GetItem()->Print("\n");
28
29      }while(pole->Advance(kontext));
30
31      pole->CloseContext(kontext);
32      pole->Close();
33
34      delete pole;
35      delete kontext;
36      delete hlavicka;
37      delete popisovac;
38
39  }
```

Výpis 1: Ukázka výpisu vektorů ze souboru perzistentního pole

Výpis z uvedeného zdrojového kódu bude podobný tomuto:

```
(34, 8, 28, 68, 3, 29),realSize: 6
(0, 51, 70, 19, 8, 18, 16, 14, 99, 17),realSize: 10
(1, 23, 71, 36, 61, 57, 77, 16),realSize: 8
(79, 72, 77, 34, 69, 12, 1, 70, 55, 41),realSize: 10
(1, 39, 80, 81, 24, 90, 94, 82, 94, 91),realSize: 10
(2, 35, 9, 81, 11, 92, 30),realSize: 7
```

Výpis 2: Výstup z metody výpisu vektorů ze souboru perzistentního pole

Na druhém příkladu si ukážeme jednoduché vkládání vektorů do souboru. Využijeme typy a objekty popsány v příkladě 1, nově nám přibyde objekt *prvek* (řádek č. 20), který zde představuje vektor ukládaný do perzistentního pole. Jako parametr kontruktoru zde předávám hlavičku perzistentního pole.

Nový soubor perzistentního pole vytvoříme metodou *Create()* (řádek č. 18), kde jako parametry vložíme název souboru, který se má vytvořit a velikost vyrovnávací paměti v počtech bloků. Metodou *SetValue()* (řádek č. 24) nastavujeme jednotlivé položky vektoru, zde neznaménková celá čísla. Prvním parametrem metody je pozice ve vektoru, kterou chceme nastavit, tím druhým pak samotná hodnota. Metoda *AddItem()* (řádek č. 28) pak vytvořený prvek uloží do perzistentního pole. Tato metoda vrací index a pozici právě vloženého prvku, třetím parametrem je prvek, který chceme vložit. Pro výpis prvku slouží metoda *Print()* (řádek č. 29), kde vložený parametr plní funkci oddělovače. Stejně jako při čtení z pole je třeba jej po ukončení práce zavřít metodou *Close()* (řádek č. 33).

Tento kód vytvoří perzistentní pole o 10 prvcích, kde na nultou pozici uloží jeho pořadí od 0 a na dalších pozicích postupně ukládá druhé mocniny až do hodnoty aktuální pozice. Výsledek bude zapsán v souboru *mojepole.dat*.

```
1  void Ukazka::Zapis_do_souboru(){
2
3      typedef cNTreeTuple_NoEncodeType Type;
4      typedef cPersistentArrayNode_VarLen<Type> tNode;
5      typedef cPersistentArray_VarLen<Type, tNode> tPersistentArray;
6
7      unsigned int index, pozice;
8      cNTreeSpaceDescriptor *popisovac = new cNTreeSpaceDescriptor(10, new cUIntType());
9
10     cPersistentArrayHeader_VarLen<Type> *hlavicka =
11         new cPersistentArrayHeader_VarLen<Type>(
12             new cNTupleSizeInfo(popisovac),
13             tNode::GetNodeExtraSize(),
14             4096);
15
16     tPersistentArray  *pole = new tPersistentArray(hlavicka);
17
18     pole->Create("mojepole.dat", 8);
19
20     cNTreeTuple *prvek = new cNTreeTuple(popisovac);
21
22     for(unsigned int i=0; i<10; i++){
23         for(unsigned int j=1; j<=i; j++){
24             prvek->SetValue(0, i);
25             prvek->SetValue(j, j*j);
26         }
27
28         pole->AddItem(index, pozice, *prvek);
29         prvek->Print("\n");
30         prvek->Clear();
31     }
32
33     pole->Close();
34
35     delete prvek;
36     delete pole;
37     delete hlavicka;
38     delete popisovac;
39
40 }
```

Výpis 3: Ukázka zápisu vektorů do souboru perzistentního pole

Výpis z uvedeného zdrojového kódu bude následující:

```
(),realSize: 0
(1, 1),realSize: 2
(2, 1, 4),realSize: 3
(3, 1, 4, 9),realSize: 4
(4, 1, 4, 9, 16),realSize: 5
(5, 1, 4, 9, 16, 25),realSize: 6
```

```
(6, 1, 4, 9, 16, 25, 36),realSize: 7
(7, 1, 4, 9, 16, 25, 36, 49),realSize: 8
(8, 1, 4, 9, 16, 25, 36, 49, 64),realSize: 9
(9, 1, 4, 9, 16, 25, 36, 49, 64, 81),realSize: 10
```

Výpis 4: Výstup z metody výpisu vektorů ze souboru perzistentního pole

Jak lze vidět z předchozích případů, samotná práce s perzistentním polem je intuitivní a velice podobné práci s běžnými soubory. Při vytváření polí a potřebných typů je dobré být pozorný a případnou alokaci prvků provádět mimo často používané metody. Ušetříme tak nemalé množství paměti, jelikož C++ nemá tak propracovanou správu paměti jako modernější jazyky jako C# nebo Java. Např. opakovanou alokaci jednoho prvku v cyklu řeší alokací nového místa na haldě a paměť potřebná k alokaci jednoho prvku tak může v extrémním případě narůst na součin velikosti prvku a délky cyklu!

5 Analýza a návrh komponenty pro externí třídění prvků

5.1 Specifikace požadavků

Mým úkolem bylo navrhnout a implementovat komponentu pro třídění prvků ze struktury perzistentního pole využívající jen předem definovanou velikost hlavní paměti. Po prostudování teorie týkající se třídění dat na externím úložišti jsem musel vzít v potaz samotná data, nad kterými bude později algoritmus běžet. Jelikož jde o data s relativně velkým hodnotovým rozsahem, zvolil jsem ve své implementaci k algoritmu pro třídění dat na externím úložišti, externímu mergesortu, optimalizaci posloupnosti spotřeby, viz. kapitola 3.3. První fáze algoritmu využívá schéma načti-setřiď-ulož popsané v kapitole 3.2.1.

Celou komponentu bylo nutno navrhnout obecně, jelikož komponenta perzistentního pole je navrhnována pro množství typů, kde každý z nich rozšiřuje základní třídu typu o metody nutné ke správnému chodu konkrétního typu. Mnou navržená komponenta tedy musí umět třídít typy rozširujici základní třídu *cDataType*, jinými slovy, tříděný typ bude mou třídu parametrizovat.

Komponenta musí splňovat:

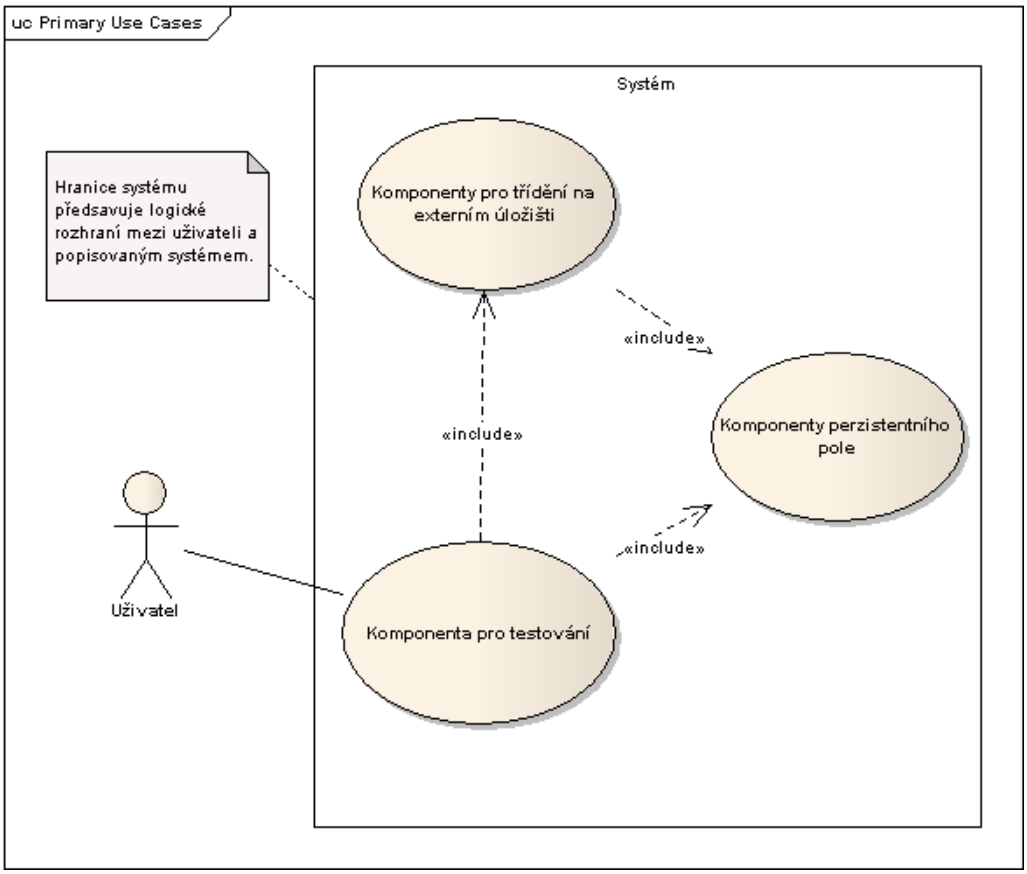
- Třídit prvky rozšiřující třídu *cDataType*, resp. jejího potomka *cBasicType*.
- Při třídění používat jen předem známou velikost hlavní paměťi.
- Komponentu optimalizovat pro reálná data, nad kterými bude běžět.
- Pro testování implementovat metodu pro načítání dat z textového souboru, dále vhodným způsobem umožnit zadávat jednotlivé parametry třídění.

Jako omezení se až v průběhu implementace ukázala být nesprávně pracující vyrovnávací pamět perzistentního pole, kvůli které bylo nutno úplně odstranit rekurzi ve druhé části mergesortu a nahradit ji upraveným nerekurzivním algoritmem. Bylo tedy nutno upravit analýzu a následnou imlementaci tak, abych se tomuto problému vyvaroval. I tak jsem se ale nevyhnul omezení v podobě nadměrně narůstajících nárocích na další hlavní paměť při běhu výsledné aplikace.

Dostupná paměť představuje pro algoritmus v první části množství hlavní paměti (v blocích definovaných při vytváření hlaviček pole, viz řádek 9 ve zdrojovém kódu 1) použitou ke třídění jednotlivých skupin. Velikost skupin v první fázi algoritmu je tedy rovna velikosti dostupné hlavní paměti. Ve druhé fázi nám velikost hlavní paměti definuje maximální počet skupin otevřených současně při běhu slévací části algotimu. Vstupní a výstupní zásobník o kterých jsme mluvili v úvodu zde nejsou potřeba, tuto režii za nás řeší samotné perzistentní pole.

5.2 Specifikace pomocí případu užití

Diagram případu užití, tzv. Use Case, nám popisuje, jak aktéři, stojící vně systému, pracují s jednotlivými funkcemi systému.



Obrázek 3: Diagram případu užití systému

V tomto případě jde o velice jednoduchý model, kdy uživatel, pracující se systémem, používá jen se samotnou komponentou pro testování třídění, která využívá třídy pro třídění na externím úložišti a třídy perzistentního pole. Komponenty pro třídění samozřejmě také využívají třídy perzistentního pole.

5.3 Třídní diagram

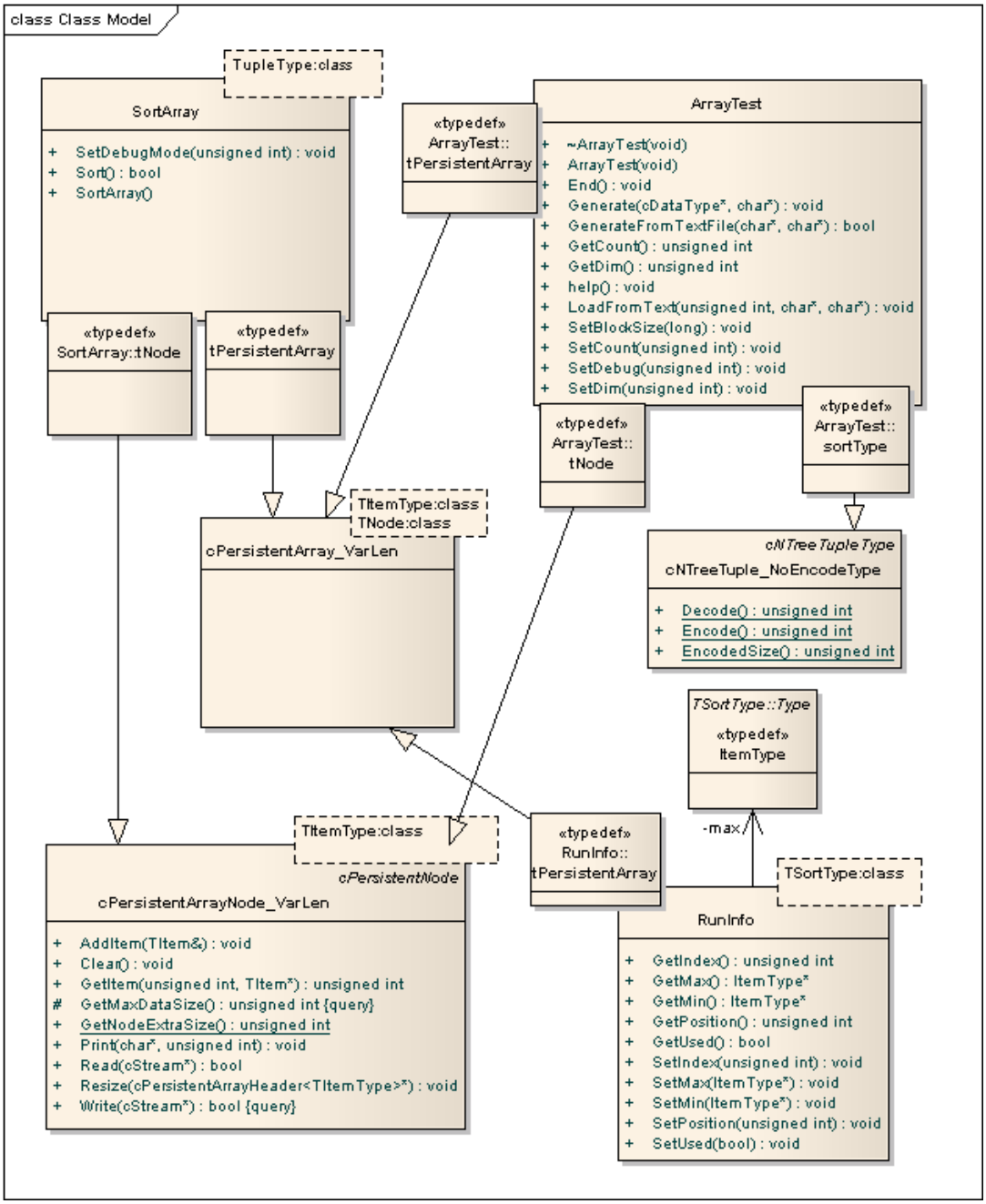
Pomocí třídního diagramu (Obrázek 4) je zde nastíněna statická struktura systému prostřednictvím jednotlivých tříd a vzájemných vztahů mezi nimi:

- *SortArray* je šablona implementující samotné třídění dat za pomocí omezeného množství hlavní paměti. Parametrizuje se konkrétní typem, představující tříděný prvek, dědící ze třídy *cBasicType*. Jsou zde jednotlivé metody pro třídění i dále popsaná optimalizace.
- šablona *RunInfo* slouží k ukládání údajů o jednotlivých setříděných skupinách vznikajících v první části algoritmu. Zaznamenává se zde index a pozice posledního

prvku skupiny, její maximální a minimální prvek a logickou hodnotu vypovídající o použití dané skupiny ve druhé části algoritmu.

- třída *ArrayTest* je třídou pro testování výše uvedených tříd, její součástí je generátor náhodných vektorů a načítání textových souborů. Uživatel s touto třídou komunikuje přes rozhraní příkazové řádky, kde zadává odpovídající parametry.
- třída *cNTreeTuple_NoEncodeType* zde představuje konkrétní tříděný prvek, jediná vazba na něj bude ze samotné testovací třídy.
- šablona *cPersistentArrayNode_VarLen* představuje vektor vkládaný do perzistentního pole, parametrizuje se konkrétním tříděným typem.
- šablona *cPersistentArray_VarLen* odpovídá již zmiňovanému konkrétnímu perzistentnímu poli. Parametrizuje se konkrétním tříděným typem a jemu odpovídajícím uzlem.

Uživatel bude s aplikací kominukovat prostřednictvím příkazové řádky, kde bude zadávat potřebné parametry. Samotná třída ArrayTest v sobě zahrnuje jak generování náhodných vektorů dle uživatelem zadaných parametrů, tak i načítání textových souborů a jejich následné převedení do perzistentního pole tak, aby danou množinu dat bylo možno dále setřídit. Setřídění perzistentního pole pak proběhne zavoláním metody *Sort()* ze třídy SortArray s odpovídajícími parametry.



Obrázek 4: Třídní diagram komponenty pro třídění prvků

6 Implementace

Cílem této kapitoly je popsat samotnou implementaci vycházející z předchozí analýzy a návrhu, popsat nejdůležitější metody komponenty pro třídění prvků na externím úložišti a jejich optimalizaci. Aplikace je naprogramována jako konzolová v jazyce C++, předávání informací a přepínačů je realizováno pomocí volitelných parametrů, kterým se podrobněji věnuji v příloze A.

V dalších kapitolách si představíme implementované třídy, jejich význam a nejdůležitější metody.

6.1 Třída ArrayTest

Třída ArrayTest slouží k otestování níže popsaných šablon pro třídění prvků na externím úložišti, obsahuje metodu *main*, která je po spuštění aplikace volána. Pro optimální testování má uživatel k dispozici jak vygenerování náhodných vektorů pole podle předem nastaveným parametrů tak možnost načtení textového souboru s jednotlivými vektory, kde každý řádek odpovídá jednomu vektoru a prvky každého vektoru jsou odděleny čárkou (dalši popis je uveden v příloze A). Po vygenerování, resp. načtení dat, jsou vektory uloženy do struktury perzistentního pole připraveny na samotné třídění. Typem tříděného prvku parametrizujeme obě potřebné šablony a voláme potřebné metody. Kromě názvu externího souboru perzistentního pole obsahující vektory ke třídění a několika zadaných parametrů, jako je počet vektorů, jejich dimenze, typ tříděného prvku a velikost bloku perzistentního pole, je nutno předat třídě *SortArray* také instanci potomka třídy *cSizeInfo* obsahující informace potřebné k výpočtu místa podle zadaných parametrů a hlavičku vytvořeného perzistentního pole.

Nyní si popíšeme některé důležité metody třídy ArrayTest. Všechny metody jsou popsány vygenerovanou programátorskou dokumentací na přiloženém CD.

6.1.1 Generování náhodných dat

Metoda *ArrayTest::Generate(cDataType *typ_prvku, char *input_file)* je určená ke generování náhodných vektorů, tedy vektorů, které mají náhodnou dimenzi od 1 do zadané maximální dimenze a jejich prvky jsou náhodně vygenerované od 0 do hodnoty globální konstanty *MAX_GEN*, nastavené standardně na 1000. Počet vektorů a jejich maximální dimenzi určují globální proměnné *count* a *dim*. V této metodě se také vytváří hlavička perzistentního pole, která bude použita jako parametr třídy pro třídění prvků.

Metoda má dva argumenty:

- prvním je typ prvku, který bude představovat jednotlivý prvek vektoru, jde o novou instanci potomka třídy *cBasicType*.
- tím druhým je název souboru perzistentního pole, který se vytvoří touto metodou.

Vkládání náhodných dat probíhá vždy pro jeden prvek, ten je uložen do struktury perzistentního pole a poté vyprázdněn a tím připraven pro další plnění.

6.1.2 Načtení dat z textového souboru

Metoda *ArrayTest::GenerateFromTextFile(char *soubor, char *vstupni_soubor)* provádí načítání vektorů uložených v textovém souboru do perzistentního pole. Nejprve jsou z prvního řádku textového souboru načteny údaje o konkrétním typu tříděných vektorů, opět jde o jednoho z potomků třídy *cBasicType*, dále o maximální dimenzi vektorů a nakonec o jejich celkovém počtu. Poté je podle načteného typu zavolána přetypovaná metoda *ArrayTest::LoadFromText()*, která zajistí správné načtení zbylých vektorů do struktury perzistentního pole.

Metoda vyžaduje dva argumenty:

- první představuje název vstupního textového souboru.
- druhý je vygenerovaný soubor struktury perzistentního pole, tedy vstupní soubor pro třídící algoritmus.

6.2 Šablona RunInfo

Jak jsme již popsali v kapitole Analýza a návrh, šablona *RunInfo* slouží k ukládání informací o jednotlivých skupinách vznikajících v první části třídícího algoritmu. Šablona je parametrizována konkrétním typem tříděného prvku a je počítáno, že jedna instance bude představovat popis jedné skupiny. Lokální proměnné *index* a *position* zaznamenávají index a pozici posledního vloženého prvku, logická hodnota *used* určuje, zda již byla tato skupina použita ve druhé části třídícího algoritmu a *max* a *min* představující maximální a minimální prvek skupiny (tyto údaje se ukládají kvůli pozdější optimalizaci). Přístup k těmto proměnným je zapouzdřen do metod *Set* a *Get*, majících standardní konvenci, tedy např. *SetIndex()* a *GetIndex()*. Pro jednodušší kopírování celých šablon je zde implementován také *operátor rovná se*.

6.3 Šablona SortArray

Nyní se konečně dostáváme k šabloně, řešící požadované třídění prvků na externím úložišti. Tato šablona se, obdobně jako šablona *RunInfo*, parametrizuje konkrétním typem tříděného prvku. Z veřejných metod obsahuje kromě konstruktoru metodu *Sort()*, jejíž zavoláním dojde k postupnému spuštění první a druhé části vnějšího třídícího algoritmu.

Obě tyto části si podrobně popíšeme.

6.3.1 Algoritmus pro vytvoření setříděných skupin

Metoda pro vytvoření setříděných skupin implementovaná podle algoritmu *načti-setřiď-ulož* popsaného v kapitole 3.2.1 se jmenuje *SortArray<TupleType>::MakeRuns(cPersistentArrayHeader_VarLen<TupleType> *mHeader, unsigned int ram_size, unsigned int count)*. Její argumenty jsou:

- hlavička perzistentního pole, kterou jsme dříve vytvořili, zde se předává hlavička z testovací třídy *ArrayTest*.
- velikost dostupné hlavní paměti pro algoritmus, je definovaná počtem volných bloků, kde jeden blok hlavní paměti odpovídá velikosti jednomu bloku perzistentního pole. Je stejná pro první i druhou fázi algoritmu. V této fázi velikost dostupné hlavní paměti určuje velikost jednotlivých setříděných skupin.
- poslední položkou je počet vektorů, které vstupní perzistentní pole obsahuje.

Metoda vrací logickou 1, pokud algoritmus proběhne korektně a v jeho průběhu nedojde k žádné chybě, logickou nulu, pokud tomu tak není.

6.3.2 Algoritmus pro slévání setříděných skupin

Metoda zajišťující slévání setříděných skupin implementovaná podle algoritmu využívajícího k-cestný mergesort, popsaný v kapitole 3.2, má název
*SortArray<TupleType>::Merge(unsigned int ram_size, char *vystupni_soubor, bool use_optimalization)*
a následující argumenty:

- velikost dostupné hlavní paměti pro algoritmus, je definovaná počtem volných bloků, kde jeden blok hlavní paměti odpovídá velikosti jednomu bloku perzistentního pole. Je stejná pro první i druhou fázi algoritmu. V této fázi velikost dostupné hlavní paměti určuje počet najednou otevřených skupin pro slévání.
- dalším parametrem je jméno výstupního souboru, jde o výsledný soubor obsahující setříděné perzistentní pole.
- posledním parametrem je logická hodnota určující, zdá má algoritmus využívat implementovanou optimalizaci.

Metoda vrací, stejně jako metoda implementující první část třídícího algoritmu, logickou 1, pokud algoritmus proběhne korektně a v jeho průběhu nedojde k žádné chybě, logickou nulu, pokud tomu tak není.

Mimo algoritmů pro samotné třídění je zde i několik pomocných metod, ať už jde o metodu *Quicksort()*, využívanou při třídění části prvků v hlavní paměti potřebnou v první fázi třídícího algoritmu, nebo metodu *Zformuj_pole_bin()*, která nově vložený prvek do pole indexů skupin binárně zařadí podle jeho velikosti na patřičné místo v poli.

6.3.3 Optimalizace algoritmu pro slévání setříděných skupin

Optimalizace implementována ve druhé části algoritmu pro třídění na externím úložišti využívá prognostickou metodu, konkrétně upravenou posloupnost spotřeby, popsanou v kapitole 3.3.

Při slévání máme k dispozici pouze N stran volné hlavní paměti, znamená to, že můžeme najednou otevřít a slévat jen N skupin. Pokud se jedna ze vkládaných skupin vyprázdní, hledáme na její místo jinou vyhovující skupinu. Nejprve z ještě neslévaných

Algoritmus 4: Pseudokód algoritmu načti-setřiď-ulož

```
Input: hlavička_vstupního_pole, velikost_paměti, počet_vektorů
vytvoř a inializuj potřebné proměnné;
vstupní_soubor.otevři();
kontext.otevři(vstupní_soubor.začátek());
výstupní_soubor.vytvoř();
počet_prvků_ve_skupině := (ram_size * block_size) / (dim * sizeof(typ));
počet_skupin := počet_vektorů / počet_prvků_ve_skupině;
if (počet_vektorů % ((int)(ram_size * block_size) / (dim * sizeof(typ))) != 0) then
    počet_skupin++;

vytvoř a inicializuj In_desc[počet_skupin+1];
//popisovač skupin
inicializuj pole[počet_prvků_ve_skupině];
//pole představující data z perzistentního pole v hlavní paměti
i:=0;
p_i:=0;
for do
    pole[i++] = kontext.vraťPrvek();
    if kontext != vstupní_soubor.konec then
        if i == počet_prvků_ve_skupině then
            Quicksort(pole, 0, i);
            for x:=0 to i do
                výstupní_soubor.přidej(pole[x]);
            Ulož informace o skupině do In_desc[p_i];
            i:=0;
            p_i++;

    else
        Quicksort(pole, 0, i);
        for x:=0 to i do
            výstupní_soubor.přidej(pole[x]);
        Ulož informace o skupině do In_desc[p_i];
        break;
kontext.zavři();
vstupní_soubor.zavři();
výstupní_soubor.zavři();
smaž již nepotřebné objekty;
```

Algoritmus 5: Pseudokód k-cestného slévání bez rekurze

```
Input: velikost_paměti, výstupní_soubor
vytvoř a inializuj potřebné proměnné;
Out_počet_skupin := 0;
for do
    vytvoř a inicializuj Out_desc[počet_skupin];
    vstup.otevři();
    výstup.vytvoř();
    vytvoř a inicializuj pole_kontextů[počet_skupin+1];
    i := 0;
    dolni := 0;
    horni := 0;
    Out_počet_skupin := 0;
    max = (velikost_paměti > počet_skupin)?počet_skupin:velikost_paměti;
    repeat
        vektor<unsigned int> pole_sk;
        for i:=dolni to horni do
            inicializuj kontext In_pole_ctx[i];
            In_pole_ctx[i].otevřiKontext(začátek ité skupiny);
            pole_sk.dejNaKonec(i);
            In_desc[i].NastavPoužití(true);
            if pole_sk.velikost() == max then
                break;
        vyvtoř a inicializuj pole_indexů_skupin;
        dolni := pole_sk[0];
        horni := pole_sk[pole_sk.velikost()-1];
        horni++;
        setřiď pole_indexů_skupin podle nejmenšího prvku In_pole_ctx;
        Slévej_skupiny();
        ulož informace o výsledné skupině do Out_desc[Out_počet_skupin];
        dolni := horni;
    until horni != počet_skupin ;
    vstup.zavři();
    výstup.zavři();
    celk_počet++;
    počet_skupin := Out_počet_skupin;
    In_desc = Out_desc;
    if Out_počet_skupin == 1 then
        break;
je-li posledním výst. souborem soubor dočasný, načti jej do souboru výst.;
smaž již nepotřebné objekty;
```

Algoritmus 6: Pseudokód metody Slévej() potřebné k algoritmu k-cestného slévání bez rekurze

```
Input: Pole_indexů_skupin, In_pole_ctx, In_desc, vstupní a výstupní perz. pole
množství_opakování = 0;
počet_sk = pole_sk.velikost();
for do
    if je-li prvek ze skupiny uložené na nultém indexu v poli pole_indexů_skupin > než
    prvek ze skupiny uložené na dalším indexu then
        binárně zatřiď prvek na nultém indexu v poli pole_indexů_skupin pomocí
        kontextu In_pole_ctx;

    výstup.přidej(prvek odpovídající In_pole_ctx[pole_indexů_skupin[0]]);
    if jde-li o první prvek vložený do skupiny then
        ulož jej jako minimum;

    if není-li prázdná skupina, ze které byl vložen poslední prvek then
        posuň kontext In_pole_ctx[pole_indexů_skupin[0]];
    else
        if má-li být použita optimalizace then
            Optimalizuj();
            if byla nalezena optimalizace then
                pokračuj ve slévání;

        else
            ulož poslední prvek jako maximum skupiny;
            zavři kontext In_pole_ctx[pole_indexů_skupin[0]];
            odeber ukončenou skupinu z pole_indexů_skupin;
            počet_sk−;
    if počet_sk == 0 then
        break;

    množství_opakování++;
```

skupin vybereme ty, které mají minimum větší nebo rovno naposledy vloženému prvku, logicky nás zajímají jen skupiny, které má cenu v aktuálním bodě dále slévat. Z těchto skupin pak vyberemu tu, jejíž maximum má nejmenší hodnotu.

Tento postup může ušetřit několik průchodů v druhé části algoritmu, v ideálním případě by slévání proběhlo jen v jednom průchodu, v nejhorším algoritmus proběhne stejně rychle, jako bez optimalizace.

Algoritmus 7: Pseudokód upravené posloupnosti spotřeby

```
ve chvíli, kdy dojde k vyprázdnění jedné ze vstupních skupin ve druhé části
algoritmu prováděj:
ind_dalšího := -1;
for pp:=horni to počet_skupin do
    if !In_desc[pp].BylaPoužita() and In_desc[pp].Minimum() >= poslední vložený prvek
    then
        if pp == -1 then
            maximum := In_desc[pp].Maximum();
            ind_dalšího := pp;
        else if In_desc[pp].Maximum() < maximum then
            ind_dalšího := pp;

if ind_dalšího != -1 then
    načti na pozici poslední vyprázdněné skupiny skupinu s indexem ind_dalšího;
    vytvoř a otevři pro ni kontext;
    In_desc[ind_dalšího].BylaPoužita(true);
```

6.4 Použité technologie a programové vybavení

- Pro vývoj komponenty používám *Visual Studio ve verzi 2008* od společnosti Microsoft. Jde o integrované vývojové prostředí pro vývoj aplikací s podporou několika programovacích jazyků, obsahuje propracovaný editor kódu podporující IntelliSense a refaktoring, výkonný debugger, grafický editor webových stránek, atd.
- Pro modelování systému pomocí jazyka UML využívám nástroje *Enterprise Architect* od společnosti Sparx Systems ve verzi 7.
- Aplikace byla vyvíjena a testována na operačním systému *Windows XP* s balíkem SP3 společnosti Microsoft.

7 Testy

V této kapitole srovnám čas běhu algoritmu s optimalizací a bez optimalizace na vhodně zvolené množině testovacích dat. Optimalizaci vycházející ze schématu posloupnosti spotřeby jsem vybral z důvodu charakteru dat, nad kterými bude algoritmus reálně běžět. Jelikož jde o částečně setříděný datový soubor s relativně velkým rozsahem, budu se snažit charakter těchto dat napodobit pro potřeby testování. Na této množině změřím časy běhu algoritmu s optimalizací a bez optimalizace pro různé velikosti hlavní paměti. Tyto testy ukáží, na kolik je zvolená optimalizace vhodná a zda přináší zlepšení ve výkonnosti algoritmu.

7.1 Data použitá pro testování

Jako základ pro jednotlivé datové kolekce používám setříděný soubor o velikosti 1 milión vektorů, kde každý z vektorů má dimenzi nejvíce 10. Prvky jednotlivých vektorů jsou nezáporná celá čísla a mají rozsah z intervalu $<0;100000>$. Tyto prvky budu záměrně vyměňovat mezi sebou podle zadaných parametrů tak, abych napodobil různě částečně setříděné soubory. Tyto parametry jsou:

- počet vyměněných prvků v rámci jednoho souboru
- vzdálenost jednotlivých vyměněných prvků

Tabulka 1 popisuje názvy a parametry souborů, které budou vstupem pro testování. Přehození i vzdálenost je uváděna relativně vzhledem k počtu vektorů v souboru, který je pro všechny testovací soubory stejný, jeden milión prvků.

Všechny soubory vycházejí ze souboru *test_setrideny*, přehazování náhodných prvků provádím pomocí jednoduchého algoritmu. V souborech *test1*, *test2* a *test3* se žádné dva prvky nepřehodí více jak jednou, tzn. pro správné setřídění těchto souborů zpět nám stačí jen např. $0,1 * 1000000$, tedy 100000 přehození. U zbývajících souborů je ponechána možnost několikanásobného přehození prvků.

Na každém testovaném souboru byla provedena série testů pro různé velikosti dostupné hlavní paměti. Pro snažší měření času třídění jsem do aplikace implementoval jednoduchý měřič, využívající metodu *clock()* knihovny *time.h*, který po skončení aplikace vypíše délku běhu třídícího algoritmu v sekundách.

Velikost hlavní paměti nabývá hodnot od 2, 5, 10 a 20 pro každý testovaný soubor, velikost bloku perzistentního pole byla ponechána na velikosti 4096. Výsledky časů testování uvádí tabulka 2, pro každý ze souborů je vygenerován graf, viz obrázky 5 - 10.

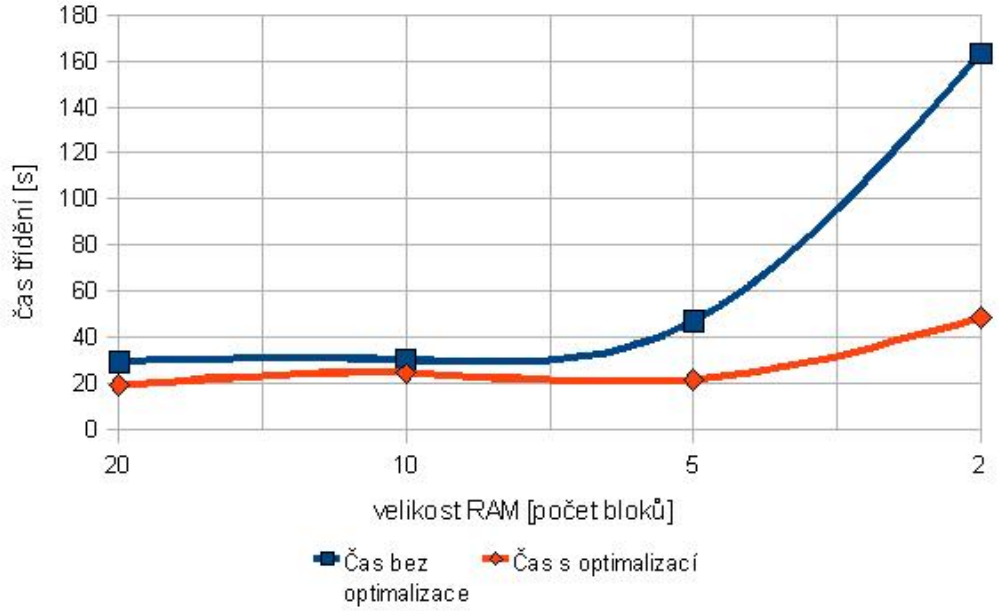
Pro správné načítání textových souborů do aplikace je nutno na prvním řádku vstupního souboru uvést hlavičku, skládající se z následujících parametrů:
TříděnýTyp; maxiální_dimenze; počet_vektorů_v_souboru. V našem případě tedy bude hlavička obsahovat *cUIntType;10;1000000*.

Název souboru	Přehození	Vzdálenost
test_setrideny.txt	0	0
test1.txt	0,1	0,1
test2.txt	0,1	0,5
test3.txt	0,1	0,8
test4.txt	0,5	0,1
test5.txt	0,8	0,1

Tabulka 1: Seznam vstupních souborů k testování a jejich parametry

Název souboru	Přehození	Vzdálenost	Velikost RAM	Čas bez op.	Čas s op.
test_setrideny.txt	0	0	20	29,4	19,2
			10	30	24,2
			5	46,9	21,5
			2	163,2	48,5
test1.txt	0,1	0,1	20	31,2	25,9
			10	39,1	35,3
			5	48,4	50
			2	123,2	112,4
test2.txt	0,1	0,5	20	32,3	25,8
			10	31,2	33,2
			5	47,4	49,9
			2	152,1	109,2
test3.txt	0,1	0,8	20	30	22,6
			10	29,6	22,3
			5	52	27,9
			2	119,3	35,9
test4.txt	0,5	0,1	20	31	22,8
			10	30,4	22,4
			5	46,5	24,5
			2	116,7	43,7
test5.txt	0,8	0,1	20	31,3	22,2
			10	31,6	23,2
			5	50,1	24,2
			2	126,4	36,1

Tabulka 2: Měření času třídění vstupních souborů pro různé velikosti hlavní paměti



Obrázek 5: Graf třídění souboru *test_setrideny.txt*

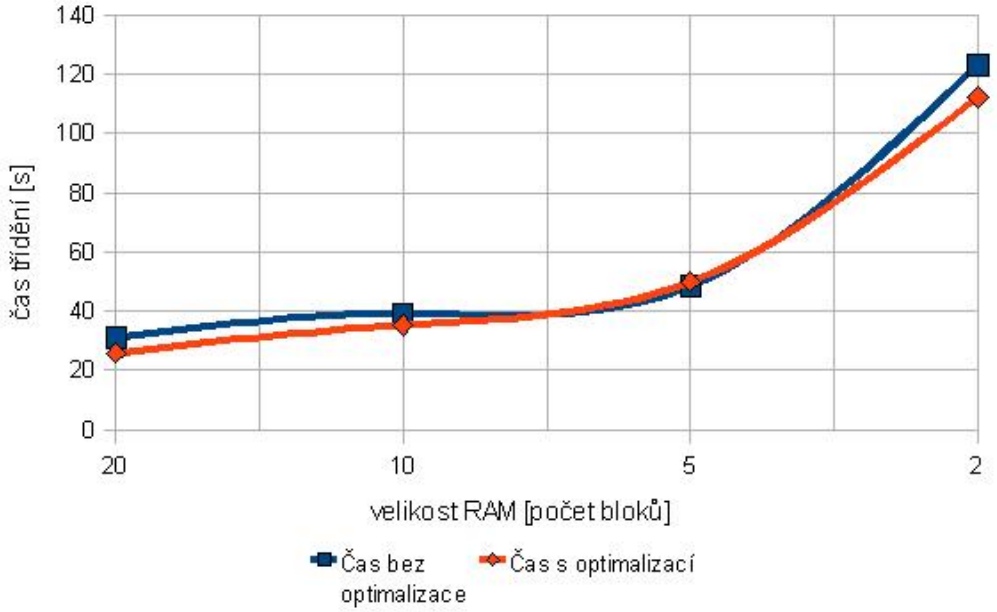
7.2 Grafy

Obrázek 5 zobrazuje třídění souboru *test_setrideny.txt*, jde o celý setříděný soubor a jak se dalo očekávat, optimalizace je zde velice účinná, v krajním případě jde o snížení doby běhu na 1/3. Spojení všech skupin proběhne díky optimalizaci pouze v jediné iteraci druhé fáze algoritmu a to pro libovolnou velikost hlavní paměti. Se snižujícím se množstvím hlavní paměti roste doba běhu algoritmu bez optimalizace, počet průchodů se totiž zvyšuje.

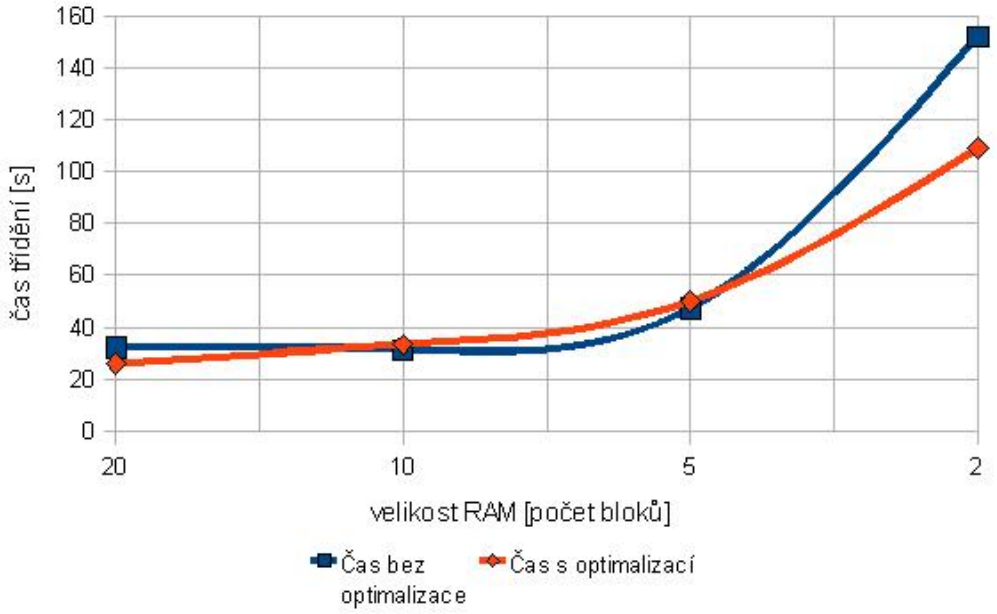
Obrázek 6 zobrazuje třídění souboru *test1.txt*, tento soubor má oproti úplně setříděnému souboru přehozeno náhodně vybraných 100 tisíc prvků ve vzdálenosti 100 tisíc. Optimalizace přináší mírné navýšení výkonnosti, kromě případu, kdy má hlavní paměť velikost 5. Jde o prodlevu způsobenou režií v podobě vyhledávání další vhodé skupiny, která nakonec není nalezena, nebo je počet celkově nalezených skupin tak malý, že optimalizace nepřinese ani takovou časovou úsporu, aby vyvážila svou režii.

Obrázek 7 zobrazuje třídění souboru *test2.txt*, tento soubor má oproti úplně setříděnému souboru přehozeno náhodně vybraných 100 tisíc prvků ve vzdálenosti 500 tisíc. Situace je zde obdobná jako u předchozího připadu, čas běhu s optimalizací kvůli nutné režii ve dvou případech překročí o několik procent čas běhu bez optimalizace.

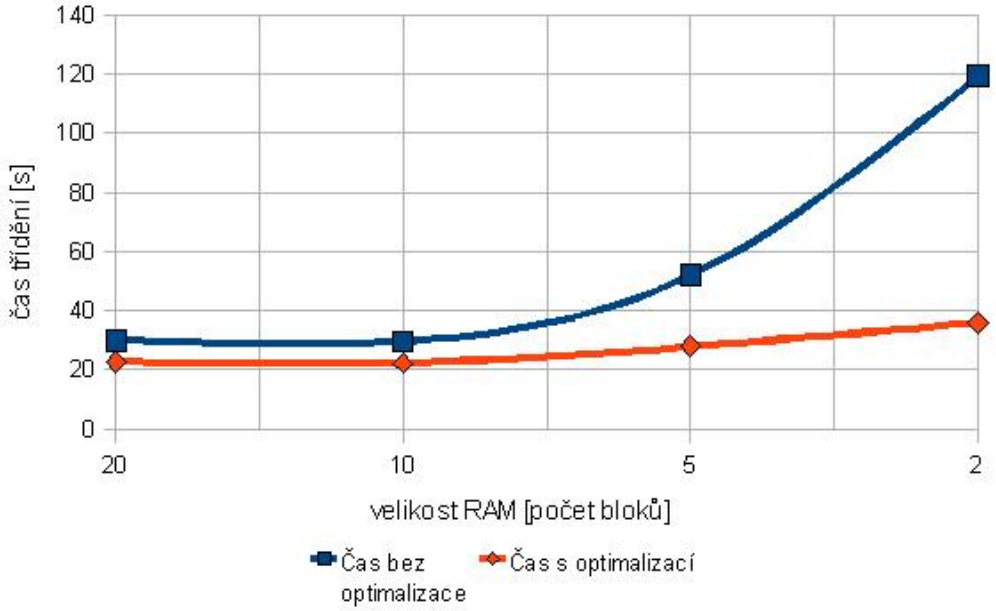
Obrázek 8 zobrazuje třídění souboru *test3.txt*, tento soubor má oproti úplně setříděnému souboru přehozeno náhodně vybraných 100 tisíc prvků ve vzdálenosti 800 tisíc. V tomto případě je optimalizace účinná, čas běhu algoritmu bez optimalizace je pro malé hodnoty hlavní paměti zhruba trojnásobný.



Obrázek 6: Graf třídění souboru *test1.txt*



Obrázek 7: Graf třídění souboru *test2.txt*



Obrázek 8: Graf třídění souboru *test3.txt*

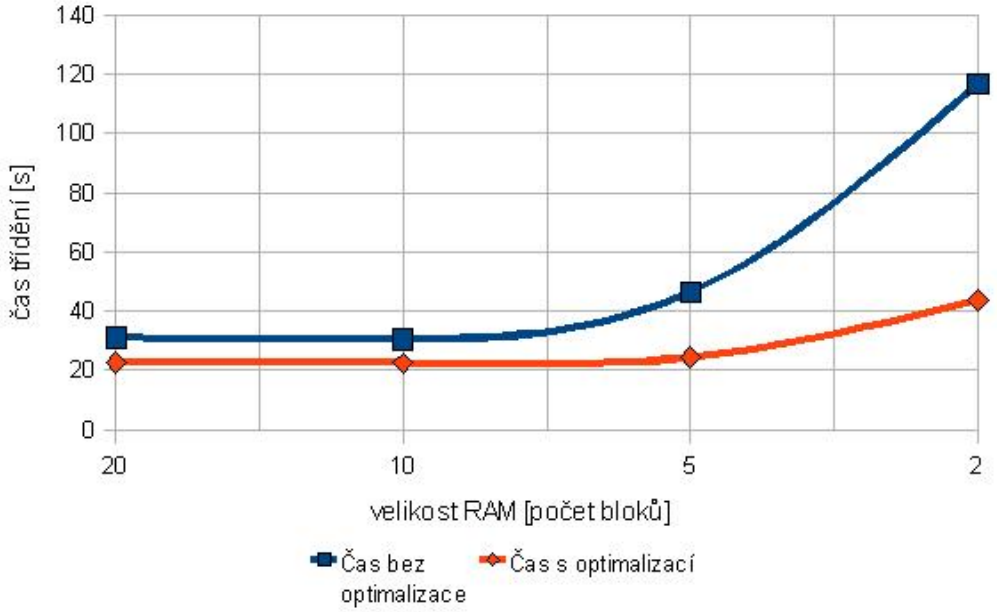
Obrázek 9 zobrazuje třídění souboru *test4.txt*, tento soubor má oproti úplně setříděnému souboru přehozeno náhodně vybraných 500 tisíc prvků ve vzdálenosti 100 tisíc. I zde je optimalizace účinná, máme-li opět k dispozici malé množství hlavní paměti. Algoritmus bez optimalizace se snižujícím se množstvím hlavní paměti znatelně ztrácí. V krajním případě nám optimalizace ušetří více jak polovinu času nutného k setřídění.

Obrázek 10 zobrazuje třídění souboru *test5.txt*, tento soubor má oproti úplně setříděnému souboru přehozeno náhodně vybraných 800 tisíc prvků ve vzdálenosti 100 tisíc. Ani nyní tomu není jinak, s klesajícím množstvím hlavní paměti vzrůstá úspora optimalizovaného algoritmu.

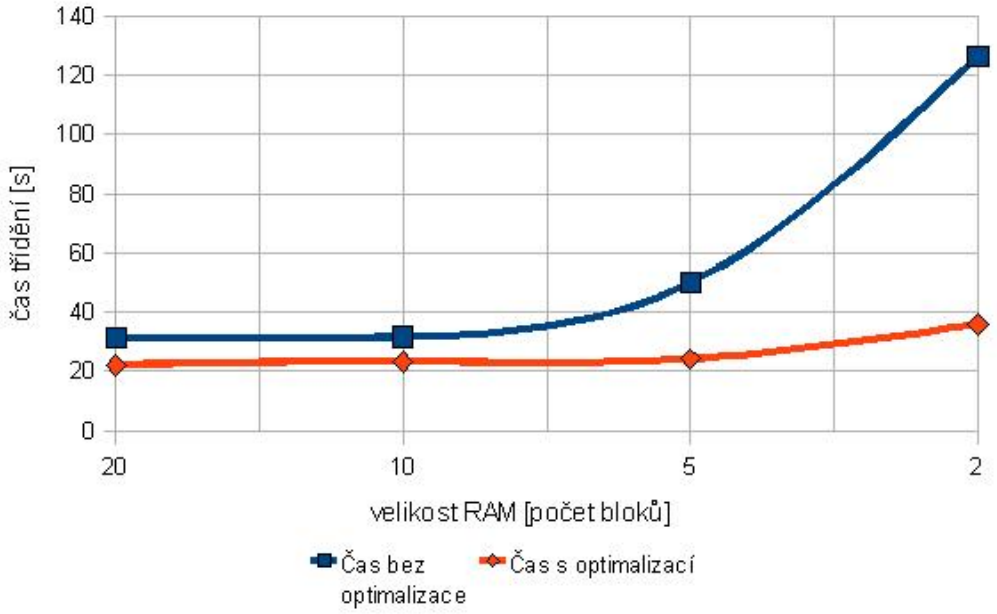
7.3 Shrnutí testování

Provedené testování potvrdilo, že optimalizace ušetří čas potřebný k setřídění, v ideálním případě setříděného souboru proběhne celé slévání v jediné iteraci. Pro více nesetříděné soubory není účinnost optimalizace výrazně větší, než u algoritmu bez optimalizace. Pokud však budeme snižovat velikost hlavní paměti, bude algoritmus bez optimalizace využívat více průchodů než algoritmus s optimalizací a účinnost optimalizace poroste.

Nejlepších výsledků bude algoritmus s optimalizací v porovnání s algoritmem bez optimalizace dosahovat, budou-li tříděné soubory částečně předtříděné a budeme-li mít k dispozici malé množství hlavní paměti.



Obrázek 9: Graf třídění souboru *test4.txt*



Obrázek 10: Graf třídění souboru *test5.txt*

Testování probíhalo na sestavě:

- Procesor AMD Athlon XP (Barton) 2000Mhz, FSB 400Mhz
- Základní deska Asus A7N8X-E Deluxe, čipová sada nForce2 Ultra 400
- Operační paměť 1GB DDR1, PC3200, 200Mhz, dualchannel
- Pevný disk Seagate Barracuda 7200.11 ST350032 0AS, 500GB, 32MB buffer, zapojený v SATA-I, nesystémový, NTFS, velikost alokační jednotky 4kB
- Operační systém Windows XP s balíkem SP3 a posledními aktualizacemi

8 Závěr

V této práci jsou detailně popsány nejpoužívanější algoritmy pro vnější třídění a jejich nejznámější optimalizace. Po důkladném seznámení se s popsanými třídícími algoritmy jsem pro třídění dat na externím úložišti použil výkonný algoritmus externí mergesort. Na základě tohoto algoritmu jsem navrhl a implementoval v jazyce C++ třídící komponentu, která řeší třídění dat z perzistentního pole za použití předem definovaného množství hlavní paměti a pro níž byla navržena a implementována optimalizace. Komponenta využívá třídy perzistentního pole implemetované databázovou skupinou na Katedře informatiky Fakulty elektrotechniky a informatiky, Vysoké školy Báňské - Technické univerzity Ostrava. Oproti původním předpokladům musela analýza i implementace komponenty doznat několika změn, z důvodu omezených možností použitého prezistního pole, které nemožňovalo použití rekurze v druhé části mergesortu. Nad celou implementací byla provedena série testů z nichž lze vyvodit, že implementovaná optimalizace je zvláště výkonná v případě velmi setříděných souborů a dále v případech, kdy je množství hlavní paměti malé. Optimalizace se pro tento případ projevila jako vhodná, jelikož reálná data, nad kterými bude algorimus běžet, budou částečně setříděná.

Popis ovládání celé komponenty je uveden v uživatelské příručce, která je přílohou této práce. Popis implementovaných metod a jejich parametrů je uveden v generované dokumentaci, která je součástí přiloženého CD.

Z výše uvedeného vyplývá, že cíl diplomové práce byl splněn.

9 Reference

[1] *MANOLOPOULOS, Yannis, THEODORIDIS, Yannis, VASSILIS J., Tsotras. Advanced Database Indexing. 2000th edition. Boston : Kluwer Academic Publishers, 2000. 286 s. Kluwer international series on advances in database systems; sv. 17. ISBN 0792377168.*

[2] KRÁTKÝ, Michal, DVORSKÝ, Jiří. *Úvod do programování. Úvod do programování, 2004/2005 [online]. 2004/2005 [cit. 2009-10-20]. Dostupný z WWW: <http://www.cs.vsb.cz/kratky/courses/2004-05/udp/presentation/udp-1_6.pdf>.*

[3] *KNUTH, Donald Ervin. The Art Of Computer Programming : Volume 3: Sorting and Searching. Druhé vydání. Massachusetts : Addison-Wesley Professional, 1998. 800 s. ISBN 0201896850.*

[4] *Intel X18-M/X25-M SATA Solid State Drive 34nm Product Line [online]. Leden 2010. Intel, 2010 [cit. 2010-07-10]. Dostupné z WWW: <http://download.intel.com/design/flash/nand/mainstream/322296.pdf>.*

[5] *ESTIVILL-CASTRO, Vladimir, WOOD, Derick. Foundations for Faster External Sorting. Volume 880. Heidelberg : Springer Berlin, 1994. ISBN 9783540587156.*

[6] *ZHANG, Weiye, LARSON, Per-Ake. Buffering and read-ahead strategies for external mergesort. In Proceedings of the Conference on Very Large Databases (VLDB). San Francisco : Morgan Kaufmann Publishers Inc. 1998. 523–533. ISBN 1558605665.*

[7] *KAGEL, Art S. National Institute of Standards and Technology [online]. 16.5.2005 [cit. 2010-06-23]. Balanced two-way merge sort. Dostupné z WWW: <http://www.itl.nist.gov/div897/sqg/dads/HTML/balanc2wayms.html>.*

A Uživatelská příručka

V této kapitole Vás seznámím s prací s aplikací pro testování třídění dat na externím úložišti. Implementovaná aplikace je konzolová, spouští se přikazem *ArrayTest_VarLen.exe* ze složky *\source\test\PersistenArray\Debug* na přiloženém CD a ovládá se následujícími parametry:

- -o nastaví jméno výstupního souboru, výchozí hodnota je Output.dat
- -O nasdtaví použití optimalizace
- -d dimenze vektoru, výchozí hodnota 10
- -c počet vektorů, výchozí hodnota 50000
- -r velikost RAM (v blocích), výchozí hodnota 20
- -b velikost bloku, výchozí hodnota 4096
- -h vypíše nápovědu
- -l aktivuje ladící mod s jednoduchým výpisem
- -L aktivuje ladící mod s rozšířeným výpisem
- -N soubor.txt načte vstupní data z textového souboru

Spustí-li uživatel aplikaci bez parametrů, vypíše aplikace jednoduchou nápovědu se seznamem jednotlivých argumentů a jejich popisem.

Nyní několik jednoduchých příkladů:

- *ArrayTest_VarLen.exe -c 500000 -d 10 -r 4 -l -O*
  vygeneruje a setřídí perzistentní pole o velikosti 500000 vektorů maximální dimenze 10 s velikostí bloku 4096 a s použitím 4 bloků hlavní paměti. Při třídění bude využita optimalizace a aplikace bude poskytovat základní výpisy. Setříděný soubor bude uložen ve výchozím souboru *Output.dat*.

- *ArrayTest_VarLen.exe -N test_setrideny.txt -l -r 10 -o Mujvystup.dat*
  načte a setřídí textový soubor *test_setrideny.txt*. Při třídění bude využívat 10 bloků hlavní paměti, perzistentní pole bude mít výchozí hodnotu velikosti bloku, tedy 4096. Pri třídění se budou zobrazovat základní výpisy, setříděný soubor bude uložen v souboru *Mujvystup.dat*.

- *ArrayTest_VarLen.exe -N test1.txt*
  tento příkaz načte a setřídí textový soubor *test1.txt*, při třídění bude používat 20 bloků hlavní paměti, velikost bloku perzistentního pole bude opět standardních 4096. Setříděný výstup bude uložen v souboru *Output.dat* a při třídění budou zobrazovány jen velice omezené výpisy.

B Obsah přiloženého CD

- *docs* - elektronická verze diplomové práce ve formátech Pdf a TEX.
- *prog_documentation* - generovaná programátorská dokumentace.
- *source* - zdrojové kódy komponenty a aplikace
- *source**test**PersistenArray**Debug*\ - adresář se spustitelnou aplikací pro testování třídění včetně textových dat používaných v kapitole 7.
- *source**test**PersistenArray**VariableLength*\ - adresář se zdrojovými kódy komponenty pro třídění prvků na externím úložišti.